

Java NIO

简明教程

作者：Jakob Jenkov　　翻译：小文子

目錄

Java NIO 简明教程

在查阅NIO相关资料时，发现一份很不错的资源，一位老外写的nio系列教程：

> http://tutorials.jenkov.com/java-nio/index.html

通读的过程中发现作者的文笔非常好，把技术概念讲的透彻，浅显易懂。
教程质量整体非常不错 ，故而将其翻译为中文版😎。

翻译系列暂定名为《Java NIO 简明教程》。

版本	修订时间
v1.0	2016/05/11
v1.1	2016/07/07
v1.2	2017/11/01

在线阅读地址：http://nio.hacktons.cn/

联系

- 作者：info@jenkov.com
- 译者：chaobinwu89@gmail.com @小文子

01. Java NIO 教程

> 原文链接：http://tutorials.jenkov.com/java-nio/index.html

- Java NIO: Channels and Buffers
- Java NIO: Non-blocking IO
- Java NIO: Selectors

Java NIO是java 1.4之后新出的一套IO接口，这里的的新是相对于原有标准的Java IO和Java Networking接口。NIO提供了一种完全不同的操作方式。

> NIO中的N可以理解为Non-blocking，不单纯是New

Java NIO: Channels and Buffers

标准的IO编程接口是面向字节流和字符流的。而NIO是面向通道和缓冲区的，数据总是从通道中读到buffer缓冲区内，或者从buffer写入到通道中。

Java NIO: Non-blocking IO

Java NIO使我们可以进行非阻塞IO操作。比如说，单线程中从通道读取数据到buffer，同时可以继续做别的事情，当数据读取到buffer中后，线程再继续处理数据。写数据也是一样的。

Java NIO: Selectors

NIO中有一个“slectors”的概念。selector可以检测多个通道的事件状态（例如：链接打开，数据到达）这样单线程就可以操作多个通道的数据。 所有这些都会在后续章节中更详细的介绍。

02. Java NIO 概览

> 原文链接：http://tutorials.jenkov.com/java-nio/overview.html

NIO包含下面几个核心的组件：

- Channels
- Buffers
- Selectors

整个NIO体系包含的类远远不止这几个，但是在笔者看来Channel,Buffer和Selector组成了这个核心的API。其他的一些组件，比如Pipe和FileLock仅仅只作为上述三个的负责类。因此在概览这一节中，会重点关注这三个概念。其他的组件会在各自的部分单独介绍。

通道和缓冲区（Channels and Buffers）

通常来说NIO中的所有IO都是从Channel开始的。Channel和流有点类似。通过Channel，我们即可以从Channel把数据写到Buffer中，也可以把数据冲Buffer写入到Channel，下图是一个示意图：



Java NIO: Channels read data into Buffers, and Buffers write data into Channels

有很多的Channel，Buffer类型。下面列举了主要的几种：

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

正如你看到的，这些channel基于于UDP和TCP的网络IO，以及文件IO。 和这些类一起的还有其他一些比较有趣的接口，在本节中暂时不多介绍。为了简洁起见，我们会在必要的时候引入这些概念。 下面是核心的Buffer实现类的列表：

- ByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer

- IntBuffer
- LongBuffer
- ShortBuffer

这些Buffer涵盖了可以通过IO操作的基础类型：byte,short,int,long,float,double以及characters. NIO实际上还包含一种MappedBytesBuffer,一般用于和内存映射的文件。

选择器（Selectors）

选择器允许单线程操作多个通道。如果你的程序中有大量的链接，同时每个链接的IO带宽不高的话，这个特性将会非常有帮助。比如聊天服务器。 下面是一个单线程中Slector维护3个Channel的示意图：



Java NIO: A Thread uses a Selector to handle 3 Channel's

要使用Selector的话，我们必须把Channel注册到Selector上，然后就可以调用Selector的select()方法。这个方法会进入阻塞，直到有一个channel的状态符合条件。当方法返回后，线程可以处理这些事件。

03. Java NIO Channel通道

> 原文链接：http://tutorials.jenkov.com/java-nio/channels.html

- Channel的实现（Channel Implementations）
- Channel的基础示例（Basic Channel Example）

Java NIO Channel通道和流非常相似，主要有以下几点区别：

- 通道可以读也可以写，流一般来说是单向的（只能读或者写）。
- 通道可以异步读写。
- 通道总是基于缓冲区Buffer来读写。

正如上面提到的，我们可以从通道中读取数据，写入到buffer；也可以中buffer内读数据，写入到通道中。下面有个示意图：



Java NIO: Channels read data into Buffers, and Buffers write data into Channels

Channel的实现（Channel Implementations）

下面列出Java NIO中最重要的集中Channel的实现：

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

FileChannel用于文件的数据读写。 DatagramChannel用于UDP的数据读写。 SocketChannel用于TCP的数据读写。 ServerSocketChannel允许我们监听TCP链接请求，每个请求会创建会一个SocketChannel.

Channel的基础示例（Basic Channel Example）

这有一个利用FileChannel读取数据到Buffer的例子：

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {

  System.out.println("Read " + bytesRead);
  buf.flip();

  while(buf.hasRemaining()){
      System.out.print((char) buf.get());
  }

  buf.clear();
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

注意buf.flip()的调用。首先把数据读取到Buffer中，然后调用flip()方法。接着再把数据读取出来。在后续的章节中我们还会讲解先关知识。

04. Java NIO Buffer缓冲区

> 原文链接：http://tutorials.jenkov.com/java-nio/buffers.html

Java NIO Buffers用于和NIO Channel交互。正如你已经知道的，我们从channel中读取数据到buffers里，从buffer把数据写入到channels.

buffer本质上就是一块内存区，可以用来写入数据，并在稍后读取出来。这块内存被NIO Buffer包裹起来，对外提供一系列的读写方便开发的接口。

Buffer基本用法（Basic Buffer Usage）

利用Buffer读写数据，通常遵循四个步骤：

- 把数据写入buffer；
- 调用flip；
- 从Buffer中读取数据；
- 调用buffer.clear()或者buffer.compact()

当写入数据到buffer中时，buffer会记录已经写入的数据大小。当需要读数据时，通过flip()方法把buffer从写模式调整为读模式；在读模式下，可以读取所有已经写入的数据。

当读取完数据后，需要清空buffer，以满足后续写入操作。清空buffer有两种方式：调用clear()或compact()方法。clear会清空整个buffer，compact则只清空已读取的数据，未被读取的数据会被移动到buffer的开始位置，写入位置则近跟着未读数据之后。

这里有一个简单的buffer案例，包括了write，flip和clear操作：

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();

//create buffer with capacity of 48 bytes
ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf); //read into buffer.
while (bytesRead != -1) {

  buf.flip();  //make buffer ready for read

  while(buf.hasRemaining()){
      System.out.print((char) buf.get()); // read 1 byte at a time
  }

  buf.clear(); //make buffer ready for writing
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

Buffer的容量，位置，上限（Buffer Capacity, Position and Limit）

buffer缓冲区实质上就是一块内存，用于写入数据，也供后续再次读取数据。这块内存被NIO Buffer管理，并提供一系列的方法用于更简单的操作这块内存。

一个Buffer有三个属性是必须掌握的，分别是：

- capacity容量
- position位置
- limit限制

position和limit的具体含义取决于当前buffer的模式。capacity在两种模式下都表示容量。

下面有张示例图，描诉了不同模式下position和limit的含义：



Buffer capacity, position and limit in write and read mode.

容量（Capacity）

作为一块内存，buffer有一个固定的大小，叫做capacity容量。也就是最多只能写入容量值得字节，整形等数据。一旦buffer写满了就需要清空已读数据以便下次继续写入新的数据。

位置（Position）

当写入数据到Buffer的时候需要中一个确定的位置开始，默认初始化时这个位置position为0，一旦写入了数据比如一个字节，整形数据，那么position的值就会指向数据之后的一个单元，position最大可以到capacity-1.

当从Buffer读取数据时，也需要从一个确定的位置开始。buffer从写入模式变为读取模式时，position会归零，每次读取后，position向后移动。

上限（Limit）

在写模式，limit的含义是我们所能写入的最大数据量。它等同于buffer的容量。

一旦切换到读模式，limit则代表我们所能读取的最大数据量，他的值等同于写模式下position的位置。

数据读取的上限时buffer中已有的数据，也就是limit的位置（原position所指的位置）。

Buffer Types

Java NIO有如下具体的Buffer类型：

- ByteBuffer
- MappedByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

正如你看到的，Buffer的类型代表了不同数据类型，换句话说，Buffer中的数据可以是上述的基本类型；

MappedByteBuffer稍有不同，我们会单独介绍。

分配一个Buffer（Allocating a Buffer）

为了获取一个Buffer对象，你必须先分配。每个Buffer实现类都有一个allocate()方法用于分配内存。下面看一个实例,开辟一个48字节大小的buffer：

```
ByteBuffer buf = ByteBuffer.allocate(48);
```

开辟一个1024个字符的CharBuffer：

```
CharBuffer buf = CharBuffer.allocate(1024);
```

写入数据到Buffer（Writing Data to a Buffer）

写数据到Buffer有两种方法：

- 从Channel中写数据到Buffer
- 手动写数据到Buffer，调用put方法

下面是一个实例，演示从Channel写数据到Buffer：

```
 int bytesRead = inChannel.read(buf); //read into buffer.
```

通过put写数据：

```
buf.put(127);
```

put方法有很多不同版本，对应不同的写数据方法。例如把数据写到特定的位置，或者把一个字节数据写入buffer。看考JavaDoc文档可以查阅的更多数据。

翻转（flip()）

flip()方法可以吧Buffer从写模式切换到读模式。调用flip方法会把position归零，并设置limit为之前的position的值。 也就是说，现在position代表的是读取位置，limit标示的是已写入的数据位置。

从Buffer读取数据（Reading Data from a Buffer）

冲Buffer读数据也有两种方式。

- 从buffer读数据到channel
- 从buffer直接读取数据，调用get方法

读取数据到channel的例子：

```
//read from buffer into channel.
int bytesWritten = inChannel.write(buf);
```

调用get读取数据的例子：

```
byte aByte = buf.get();
```

get也有诸多版本，对应了不同的读取方式。

rewind()

Buffer.rewind()方法将position置为0，这样我们可以重复读取buffer中的数据。limit保持不变。

clear() and compact()

一旦我们从buffer中读取完数据，需要复用buffer为下次写数据做准备。只需要调用clear或compact方法。

clear方法会重置position为0，limit为capacity，也就是整个Buffer清空。实际上Buffer中数据并没有清空，我们只是把标记为修改了。

如果Buffer还有一些数据没有读取完，调用clear就会导致这部分数据被“遗忘”，因为我们没有标记这部分数据未读。

针对这种情况，如果需要保留未读数据，那么可以使用compact。 因此compact和clear的区别就在于对未读数据的处理，是保留这部分数据还是一起清空。

mark() and reset()

通过mark方法可以标记当前的position，通过reset来恢复mark的位置，这个非常像canva的save和restore：

```
buffer.mark();

//call buffer.get() a couple of times, e.g. during parsing.

buffer.reset();  //set position back to mark.
```

equals() and compareTo()

可以用eqauls和compareTo比较两个buffer

equals()

判断两个buffer相对，需满足：

- 类型相同
- buffer中剩余字节数相同
- 所有剩余字节相等

从上面的三个条件可以看出，equals只比较buffer中的部分内容，并不会去比较每一个元素。

compareTo()

compareTo也是比较buffer中的剩余元素，只不过这个方法适用于比较排序的：

05. Java NIO Scatter / Gather

> 原文链接：http://tutorials.jenkov.com/java-nio/scatter-gather.html

- Scattering Reads
- Gathering Writes

Java NIO发布时内置了对scatter / gather的支持。scatter / gather是通过通道读写数据的两个概念。

Scattering read指的是从通道读取的操作能把数据写入多个buffer，也就是sctters代表了数据从一个channel到多个buffer的过程。

gathering write则正好相反，表示的是从多个buffer把数据写入到一个channel中。

Scatter/gather在有些场景下会非常有用，比如需要处理多份分开传输的数据。举例来说，假设一个消息包含了header和body，我们可能会把header和body保存在不同独立buffer中，这种分开处理header与body的做法会使开发更简明。

Scattering Reads

"scattering read"是把数据从单个Channel写入到多个buffer，下面是示意图：



Java NIO: Scattering Read

用代码来表示的话如下：

```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);

ByteBuffer[] bufferArray = { header, body };

channel.read(bufferArray);
```

观察代码可以发现，我们把多个buffer写在了一个数组中，然后把数组传递给channel.read()方法。read()方法内部会负责把数据按顺序写进传入的buffer数组内。一个buffer写满后，接着写到下一个buffer中。

实际上，scattering read内部必须写满一个buffer后才会向后移动到下一个buffer，因此这并不适合消息大小会动态改变的部分，也就是说，如果你有一个header和body，并且header有一个固定的大小（比如128字节）,这种情形下可以正常工作。

Gathering Writes

"gathering write"把多个buffer的数据写入到同一个channel中，下面是示意图：



Java NIO: Gathering Write

用代码表示的话如下：

```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);

//write data into buffers

ByteBuffer[] bufferArray = { header, body };

channel.write(bufferArray);
```

类似的传入一个buffer数组给write，内部机会按顺序将数组内的内容写进channel，这里需要注意，写入的时候针对的是buffer中position到limit之间的数据。也就是如果buffer的容量是128字节，但它只包含了58字节数据，那么写入的时候只有58字节会真正写入。因此gathering write是可以适用于可变大小的message的，这和scattering reads不同。

06. Java NIO Channel to Channel Transfers 通道传输接口

> 原文链接：http://tutorials.jenkov.com/java-nio/channel-to-channel-transfers.html

- transferFrom())
- transferTo())

在Java NIO中如果一个channel是FileChannel类型的，那么他可以直接把数据传输到另一个channel。逐个特性得益于FileChannel包含的transferTo和transferFrom两个方法。

transferFrom()

FileChannel.transferFrom方法把数据从通道源传输到FileChannel：

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();

long position = 0;
long count    = fromChannel.size();

toChannel.transferFrom(fromChannel, position, count);
```

transferFrom的参数position和count表示目标文件的写入位置和最多写入的数据量。如果通道源的数据小于count那么就传实际有的数据量。 另外，有些SocketChannel的实现在传输时只会传输哪些处于就绪状态的数据，即使SocketChannel后续会有更多可用数据。因此，这个传输过程可能不会传输整个的数据。

transferTo()

transferTo方法把FileChannel数据传输到另一个channel,下面是案例：

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();

long position = 0;
long count    = fromChannel.size();

fromChannel.transferTo(position, count, toChannel);
```

这段代码和之前介绍transfer时的代码非常相似，区别只在于调用方法的是哪个FileChannel.

SocketChannel的问题也存在与transferTo.SocketChannel的实现可能只在发送的buffer填充满后才发送，并结束。

07. Java NIO Selector选择器

> 原文链接：http://tutorials.jenkov.com/java-nio/selectors.html

Selector是Java NIO中的一个组件，用于检查一个或多个NIO Channel的状态是否处于可读、可写。如此可以实现单线程管理多个channels,也就是可以管理多个网络链接。

为什么使用Selector（Why Use a Selector?）

用单线程处理多个channels的好处是我需要更少的线程来处理channel。实际上，你甚至可以用一个线程来处理所有的channels。从操作系统的角度来看，切换线程开销是比较昂贵的，并且每个线程都需要占用系统资源，因此暂用线程越少越好。

需要留意的是，现代操作系统和CPU在多任务处理上已经变得越来越好，所以多线程带来的影响也越来越小。如果一个CPU是多核的，如果不执行多任务反而是浪费了机器的性能。不过这些设计讨论是另外的话题了。简而言之，通过Selector我们可以实现单线程操作多个channel。

这有一幅示意图，描述了单线程处理三个channel的情况：



Java NIO: A Thread uses a Selector to handle 3 Channel's

创建Selector(Creating a Selector)

创建一个Selector可以通过Selector.open()方法：

```
Selector selector = Selector.open();
```

注册Channel到Selector上(Registering Channels with the Selector)

为了同Selector挂了Channel，我们必须先把Channel注册到Selector上，这个操作使用SelectableChannel。register()：

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

Channel必须是非阻塞的。所以FileChannel不适用Selector，因为FileChannel不能切换为非阻塞模式。Socket channel可以正常使用。

注意register的第二个参数，这个参数是一个"关注集合"，代表我们关注的channel状态，有四种基础类型可供监听：

1. Connect
2. Accept
3. Read
4. Write

一个channel触发了一个事件也可视作该事件处于就绪状态。因此当channel与server连接成功后，那么就是"连接就绪"状态。server channel接收请求连接时处于"可连接就绪"状态。channel有数据可读时处于"读就绪"状态。channel可以进行数据写入时处于"写就绪"状态。

上述的四种就绪状态用SelectionKey中的常量表示如下：

1. SelectionKey.OP_CONNECT
2. SelectionKey.OP_ACCEPT
3. SelectionKey.OP_READ
4. SelectionKey.OP_WRITE

如果对多个事件感兴趣可利用位的或运算结合多个常量，比如：

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

SelectionKey's

在上一小节中，我们利用register方法把Channel注册到了Selectors上，这个方法的返回值是SelectionKeys，这个返回的对象包含了一些比较有价值的属性：

- The interest set
- The ready set
- The Channel
- The Selector
- An attached object (optional)

这5个属性都代表什么含义呢？下面会一一介绍。

Interest Set

这个“关注集合”实际上就是我们希望处理的事件的集合，它的值就是注册时传入的参数，我们可以用按为与运算把每个事件取出来：

```
int interestSet = selectionKey.interestOps();

boolean isInterestedInAccept  = interestSet & SelectionKey.OP_ACCEPT;
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInRead    = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite   = interestSet & SelectionKey.OP_WRITE;
```

Ready Set

"就绪集合"中的值是当前channel处于就绪的值，一般来说在调用了select方法后都会需要用到就绪状态，select的介绍在胡须文章中继续展开。

```
int readySet = selectionKey.readyOps();
```

从“就绪集合”中取值的操作类似月“关注集合”的操作，当然还有更简单的方法，SelectionKey提供了一系列返回值为boolean的的方法：

```
selectionKey.isAcceptable();
selectionKey.isConnectable();
selectionKey.isReadable();
selectionKey.isWritable();
```

Channel + Selector

从SelectionKey操作Channel和Selector非常简单：

```
Channel  channel  = selectionKey.channel();
Selector selector = selectionKey.selector();
```

Attaching Objects

我们可以给一个SelectionKey附加一个Object，这样做一方面可以方便我们识别某个特定的channel，同时也增加了channel相关的附加信息。例如，可以把用于channel的buffer附加到SelectionKey上：

```
selectionKey.attach(theObject);

Object attachedObj = selectionKey.attachment();
```

附加对象的操作也可以在register的时候就执行：

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ, theObject);
```

从Selector中选择channel(Selecting Channels via a Selector)

一旦我们向Selector注册了一个或多个channel后，就可以调用select来获取channel。select方法会返回所有处于就绪状态的channel。 select方法具体如下：

- int select()
- int select(long timeout)
- int selectNow()

select()方法在返回channel之前处于阻塞状态。 select(long timeout)和select做的事一样，不过他的阻塞有一个超时限制。

selectNow()不会阻塞，根据当前状态立刻返回合适的channel。

select()方法的返回值是一个int整形，代表有多少channel处于就绪了。也就是自上一次select后有多少channel进入就绪。举例来说，假设第一次调用select时正好有一个channel就绪，那么返回值是1，并且对这个channel做任何处理，接着再次调用select，此时恰好又有一个新的channel就绪，那么返回值还是1，现在我们一共有两个channel处于就绪，但是在每次调用select时只有一个channel是就绪的。

selectedKeys()

在调用select并返回了有channel就绪之后，可以通过选中的key集合来获取channel，这个操作通过调用selectedKeys()方法：

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
```

还记得在register时的操作吧，我们register后的返回值就是SelectionKey实例，也就是我们现在通过selectedKeys()方法所返回的SelectionKey。

遍历这些SelectionKey可以通过如下方法：

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

    SelectionKey key = keyIterator.next();

    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.

    } else if (key.isConnectable()) {
        // a connection was established with a remote server.

    } else if (key.isReadable()) {
        // a channel is ready for reading

    } else if (key.isWritable()) {
        // a channel is ready for writing
    }

    keyIterator.remove();
}
```

上述循环会迭代key集合，针对每个key我们单独判断他是处于何种就绪状态。

注意keyIterater.remove()方法的调用，Selector本身并不会移除SelectionKey对象，这个操作需要我们收到执行。当下次channel处于就绪是，Selector任然会吧这些key再次加入进来。

SelectionKey.channel返回的channel实例需要强转为我们实际使用的具体的channel类型，例如ServerSocketChannel或SocketChannel.

wakeUp()

y由于调用select而被阻塞的线程，可以通过调用Selector.wakeup()来唤醒即便此时已然没有channel处于就绪状态。具体操作是，在另外一个线程调用wakeup，被阻塞与select方法的线程就会立刻返回。

close()

当操作Selector完毕后，需要调用close方法。close的调用会关闭Selector并使相关的SelectionKey都无效。channel本身不管被关闭。

完整的Selector案例(Full Selector Example)

这有一个完整的案例，首先打开一个Selector,然后注册channel，最后锦亭Selector的状态：

```
Selector selector = Selector.open();

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP_READ);


while(true) {

  int readyChannels = selector.select();

  if(readyChannels == 0) continue;


  Set<SelectionKey> selectedKeys = selector.selectedKeys();

  Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

  while(keyIterator.hasNext()) {

    SelectionKey key = keyIterator.next();

    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.

    } else if (key.isConnectable()) {
        // a connection was established with a remote server.

    } else if (key.isReadable()) {
        // a channel is ready for reading

    } else if (key.isWritable()) {
        // a channel is ready for writing
    }

    keyIterator.remove();
  }
}
```

08. Java NIO FileChannel文件通道

> 原文链接：http://tutorials.jenkov.com/java-nio/file-channel.html

Java NIO中的FileChannel是用于连接文件的通道。通过文件通道可以读、写文件的数据。Java NIO的FileChannel是相对标准Java IO API的可选接口。

FileChannel不可以设置为非阻塞模式，他只能在阻塞模式下运行。

打开文件通道（Opening a FileChannel）

在使用FileChannel前必须打开通道，打开一个文件通道需要通过输入/输出流或者RandomAccessFile，下面是通过RandomAccessFile打开文件通道的案例：

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
```

从文件通道内读取数据（Reading Data from a FileChannel）

读取文件通道的数据可以通过read方法：

```
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);
```

首先开辟一个Buffer，从通道中读取的数据会写入Buffer内。接着就可以调用read方法，read的返回值代表有多少字节被写入了Buffer，返回-1则表示已经读取到文件结尾了。

向文件通道写入数据（Writing Data to a FileChannel）

写数据用write方法，入参是Buffer：

```
String newData = "New String to write to file..." + System.currentTimeMillis();

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());

buf.flip();

while(buf.hasRemaining()) {
    channel.write(buf);
}
```

注意这里的write调用写在了wihle循环汇总，这是因为write不能保证有多少数据真实被写入，因此需要循环写入直到没有更多数据。

关闭通道（Closing a FileChannel）

操作完毕后，需要把通道关闭：

```
channel.close();
```

FileChannel Position

当操作FileChannel的时候读和写都是基于特定起始位置的（position），获取当前的位置可以用FileChannel的position()方法，设置当前位置可以用带参数的position(long pos)方法。

```
long pos channel.position();

channel.position(pos +123);
```

假设我们把当前位置设置为文件结尾之后，那么当我们视图从通道中读取数据时就会发现返回值是-1，表示已经到达文件结尾了。 如果把当前位置设置为文件结尾之后，在想通道中写入数据，文件会自动扩展以便写入数据，但是这样会导致文件中出现类似空洞，即文件的一些位置是没有数据的。

FileChannel Size

size()方法可以返回FileChannel对应的文件的文件大小：

```
long fileSize = channel.size();
```

FileChannel Truncate

利用truncate方法可以截取指定长度的文件：

```
channel.truncate(1024);
```

FileChannel Force

force方法会把所有未写磁盘的数据都强制写入磁盘。这是因为在操作系统中出于性能考虑回把数据放入缓冲区，所以不能保证数据在调用write写入文件通道后就及时写到磁盘上了，除非手动调用force方法。 force方法需要一个布尔参数，代表是否把meta data也一并强制写入。

```
channel.force(true);
```

09. Java NIO SocketChannel套接字通道

> 原文链接：http://tutorials.jenkov.com/java-nio/socketchannel.html

在Java NIO体系中，SocketChannel是用于TCP网络连接的套接字接口，相当于Java网络编程中的Socket套接字接口。创建SocketChannel主要有两种方式，如下：

1. 打开一个SocketChannel并连接网络上的一台服务器。
2. 当ServerSocketChannel接收到一个连接请求时，会创建一个SocketChannel。

建立一个SocketChannel连接

打开一个SocketChannel可以这样操作：

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));
```

关闭一个SocketChannel连接

关闭一个SocketChannel只需要调用他的close方法，如下：

```
socketChannel.close();
```

从SocketChannel中读数据

从一个SocketChannel连接中读取数据，可以通过read()方法，如下：

```
ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = socketChannel.read(buf);
```

首先需要开辟一个Buffer。从SocketChannel中读取的数据将放到Buffer中。

接下来就是调用SocketChannel的read()方法.这个read()会把通道中的数据读到Buffer中。read()方法的返回值是一个int数据，代表此次有多少字节的数据被写入了Buffer中。如果返回的是-1,那么意味着通道内的数据已经读取完毕，到底了（链接关闭）。

向SocketChannel写数据

向SocketChannel中写入数据是通过write()方法，write也需要一个Buffer作为参数。下面看一下具体的示例：

```
String newData = "New String to write to file..." + System.currentTimeMillis();

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());

buf.flip();

while(buf.hasRemaining()) {
    channel.write(buf);
}
```

仔细观察代码，这里我们把write()的调用放在了while循环中。这是因为我们无法保证在write的时候实际写入了多少字节的数据，因此我们通过一个循环操作，不断把Buffer中数据写入到SocketChannel中知道Buffer中的数据全部写入为止。

非阻塞模式

我们可以吧SocketChannel设置为non-blocking（非阻塞）模式。这样的话在调用connect(), read(), write()时都是异步的。

connect()

如果我们设置了一个SocketChannel是非阻塞的，那么调用connect()后，方法会在链接建立前就直接返回。为了检查当前链接是否建立成功，我们可以调用finishConnect(),如下：

```
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));

while(! socketChannel.finishConnect() ){
    //wait, or do something else...
}
```

write()

在非阻塞模式下，调用write()方法不能确保方法返回后写入操作一定得到了执行。因此我们需要把write()调用放到循环内。这和前面在讲write()时是一样的，此处就不在代码演示。

read()

在非阻塞模式下，调用read()方法也不能确保方法返回后，确实读到了数据。因此我们需要自己检查的整型返回值，这个返回值会告诉我们实际读取了多少字节的数据。

Selector结合非阻塞模式

SocketChannel的非阻塞模式可以和Selector很好的协同工作。把一个活多个SocketChannel注册到一个Selector后，我们可以通过Selector指导哪些channels通道是处于可读，可写等等状态的。后续我们会再详细阐述如果联合使用Selector与SocketChannel。

10. Java NIO ServerSocketChannel服务端套接字通道

> 原文链接：http://tutorials.jenkov.com/java-nio/server-socket-channel.html

- 打开ServerSocketChannel
- 关闭ServerSocketChannel
- 监听链接

在Java NIO中，ServerSocketChannel是用于监听TCP链接请求的通道，正如Java网络编程中的ServerSocket一样。

ServerSocketChannel实现类位于java.nio.channels包下面。 下面是一个示例程序：

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.socket().bin(new InetSocketAddress(9999));
while(true) {
  SocketChannel socketChannel = serverSocketChannel.accept();
  //do something with socketChannel...
}
```

打开ServerSocketChannel

打开一个ServerSocketChannel我们需要调用他的open()方法，例如：

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
```

关闭ServerSocketChannel

关闭一个ServerSocketChannel我们需要调用他的close()方法，例如：

```
serverSocketChannel.close();
```

监听链接

通过调用accept()方法，我们就开始监听端口上的请求连接。当accept()返回时，他会返回一个SocketChannel连接实例，实际上accept()是阻塞操作，他会阻塞带去线程知道返回一个连接；很多时候我们是不满足于监听一个连接的，因此我们会把accept()的调用放到循环中，就像这样：

```
while(true){
    SocketChannel socketChannel = serverSocketChannel.accept();
    //do something with socketChannel...
}
```

当然我们可以在循环体内加上合适的中断逻辑，而不是单纯的在while循环中写true，以此来结束循环监听；

非阻塞模式

实际上ServerSocketChannel是可以设置为非阻塞模式的。在非阻塞模式下，调用accept()函数会立刻返回，如果当前没有请求的链接，那么返回值为空null。因此我们需要手动检查返回的SocketChannel是否为空，例如：

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

serverSocketChannel.socket().bind(new InetSocketAddress(9999));
serverSocketChannel.configureBlocking(false);

while(true){
    SocketChannel socketChannel = serverSocketChannel.accept();

    if(socketChannel != null){
        //do something with socketChannel...
    }
}
```

11. Java NIO: Non-blocking Server非阻塞服务器

> 原文链接：http://tutorials.jenkov.com/java-nio/non-blocking-server.html

现在我们已经知道了Java NIO里面那些非阻塞特性是怎么工作的，但是要设计一个非阻塞的服务仍旧比较困难。非阻塞IO相对传统的阻塞IO给开发者带来了更多的挑战。在本节非阻塞服务的讲解中，我们一起来讨论这些会面临的主要挑战，同时也会给出一些潜在的解决方案。

查找关于设计非阻塞服务的相关资料是比较难的，本文提出的解决方案也只能是基于笔者个人的工作经验，构思。如果你有其他的解决方案或者是更好的点子，那么还请不吝赐教。你可以在文章下方的评论区回复，或者可以给我发送邮件，也可以直接在Twitter上联系我。

虽然本文介绍的一些点子是为Java NIO设计的，但是我相信这些思路同样适用于其他编程语言，只要他们也存在和Selector类似结构，概念。就目前我的了解来说，这些结构底层OS提供的，所以基本上你可以运用到其他编程语言中去。

非阻塞服务-GitHub源码仓（Non-blocking Server - GitHub Repository）

为了演示本文探讨的一些技术，笔者已经在GitHub上面建立了相应的源码仓，地址如下：

https://github.com/jjenkov/java-nio-server

非阻塞IO通道（Non-blocking IO Pipelines）

非阻塞的IO管道（Non-blocking IO Pipelines）可以看做是整个非阻塞IO处理过程的链条。包括在以非阻塞形式进行的读与写操作。下面有一张插图，简单的描述了一个基础的非阻塞的IO管道（Non-blocking IO Pipelines）：



我们的组件（Component）通过Selector检查当前Channel是否有数据需要写入。此时component读入数据，并且根据输入的数据input对外提供数据输出output。这个对外的数据输出output被写到了另一个Channel中。

一个非阻塞的IO管道不必同时需要读和写数据，通常来说有些管道只需要读数据，而另一些管道则只需写数据。

上面的这幅流程图仅仅展示了一个组件。实际上一个管道可能存在多个component在处理输入数据。管道的长度取决于管道具体要做的事情。

当然一个非阻塞的IO管道他也可以同时从多个Channel中读取数据，例如同时冲多个SocketChannel中读取数据；

上面的流程图实际上被简化了，图中的Component实际上负责初始化Selector，从Channel中读取数据，而不是由Channel往Selector压如数据（push），这是简化的上图容易给人带来的误解。

非阻塞和阻塞通道比较（Non-blocking vs. Blocking IO Pipelines）

非阻塞IO管道和阻塞IO管道之间最大的区别是他们各自如何从Channel（套接字socket或文件file）读写数据。

IO管道通常直接从流中（来自于socket活file的流）读取数据，然后把数据分割为连续的消息。这个处理与我们读取流信息，用tokenizer进行解析非常相似。不同的是我们在这里会把数据流分割为更大一些的消息块。我把这个过程叫做Message Reader.下面是一张说明的插图：



一个阻塞IO管道的使用可以和输入流一样调用，每次从Channel中读取一个字节的数据，阻塞自身直到有数据可读。这个流程就是一个阻塞的Messsage Reader实现。

使用阻塞IO大大简化了Message Reader的实现成本。阻塞的Message Reader无需关注没有数据返回的情形，无需关注返回部分数据或者数据解析需要被复用的问题。

相似的，一个阻塞的Message Writer也不需要关注写入部分数据，和数据复用的问题。

阻塞IO通道的缺点（**Blocking IO Pipeline Drawbacks**）

上面提到了阻塞的Message Reader易于实现，但是阻塞也给他带了不可避免的缺点，必须为每个数据数量都分配一个单独线程。原因就在于IO接口在读取数据时在有数据返回前会一直被阻塞住。这直接导致我们无法用单线程来处理一个流没有数据返回时去读取其他的流。每当一个线程尝试去读取一个流的数据，这个线程就会被阻塞直到有数据真正返回。

如果这样的IO管道运用到服务器去处理高并发的链接请求，服务器将不得不为每一个到来的链接分配一个单独的线程。如果并发数不高比如每一时刻只有几百并发，也行不会有太大问题。一旦服务器的并发数上升到百万级别，这种设计就缺乏伸缩性。每个线程需要为堆栈分配320KB（32位JVM）到1024KB(64位JVM)的内存空间。这就是说如果有1,000,000个线程，需要1TB的内存。而这些在还没开始真正处理接收到的消息前就需要（消息处理中还需要为对象开拍内存）。

为了减少线程数，很多服务器都设计了线程池，把所有接收到的请求放到队列内，每次读取一条连接进行处理。这种设计可以用下图表示：



但是这种设计要求缓冲的连接进程发送有意义的数据。如果这些连接长时间处于非活跃的状态，那么大量非活跃的连接会阻塞线程池中的所有线程。这会导致服务器的响应速度特别慢甚至无响应。

有些服务器为了减轻这个问题，采取的操作是适当增加线程池的弹性。例如，当线程池所有线程都处于饱和时，线程池可能会自动扩容，启动更多的线程来处理事务。这个解决方案会使得服务器维护大量不活跃的链接。但是需要谨记服务器所能开辟的线程数是有限制的。所有当有1,000,000个低速的链接时，服务器还是不具备伸缩性。

基础的非阻塞通道设计（**Basic Non-blocking IO Pipeline Design**）

一个非阻塞的IO通道可以用单线程读取多个数据流。这个前提是相关的流可以切换为非阻塞模式（并不是所有流都可以以非阻塞形式操作）。在非阻塞模式下，读取一个流可能返回0个或多个字节。如果流还没有可供读取的数据那么就会返回0，其他大于1的返回都表明这是实际读取到的数据；

为了避开没有数据可读的流，我们结合Java NIO中的Selector。一个Selector可以注册多个SelectableChannel实例。当我们调用select()或selectorNow()方法时Selector会返回一个有数据可读的SelectableChannel实例。这个设计可以如下插图：



读取部分信息(Reading Partial Messages)

当我们冲SelectableChannel中读取一段数据后，我们并不知道这段数据是否是完整的一个message。因为一个数据段可能包含部分message，也就是说即可能少于一个message，也可能多一个message，正如下面这张插图所示意的那样：



要处理这种截断的message，我们会遇到两个问题：

1. 检测数据段中是否包含一个完整的message
2. 在message剩余部分获取到之前，我们如何处理不完整的message

检测完整message要求Message Reader查看数据段中的数据是否至少包含一个完整的message。如果包含一个或多个完整message，这些message可以被下发到通道中处理。查找完整message的过程是个大量重复的操作，所以这个操作必须是越快越好的。

当数据段中有一个不完整的message时，无论不完整消息是整个数据段还是说在完整message前后，这个不完整的message数据都需要在剩余部分获得前存储起来。

检查message完整性和存储不完整message都是Message Reader的职责。为了避免混淆来自不同Channel的数据，我们为每一个Channel分配一个Message Reader。整个设计大概是这样的：



当我们通过Selector获取到一个有数据可以读取的Channel之后，改Channel关联的Message Reader会读取数据，并且把数据打断为Message块。得到完整的message后就可以通过通道下发到其他组件进行处理。

一个Message Reader自然是协议相关的。他需要知道message的格式以便读取。如果我们的服务器是跨协议复用的，那他必须实现Message Reader的协议-大致类似于接收一个Message Reader工厂作为配置参数。

存储不完整的Message（Storing Partial Messages）

现在我们已经明确了由Message Reader负责不完整消息的存储直到接收到完整的消息。闲杂我们还需要知道这个存储过程需要如何来实现。

在设计的时候我们需要考虑两个关键因素：

1. 我们希望在拷贝消息数据的时候数据量能尽可能的小，拷贝量越大则性能相对越低；
2. 我们希望完整的消息是以顺序的字节存储，这样方便进行数据的解析；

为每个Message Reade分配Buffer（A Buffer Per Message Reader）

显然不完整的消息数据需要存储在某种buffer中。比较直接的办法是我们为每个Message Reader都分配一个内部的buffer成员。但是，多大的buffer才合适呢？这个buffer必须能存储下一个message最大的大小。如果一个message最大是1MB，那每个Message Reader内部的buffer就至少有1MB大小。

在百万级别的并发链接数下，1MB的buffer基本没法正常工作。举例来说，1,000,000 x 1MB就是1TB的内存大小！如果消息的最大数据量是16MB又需要多少内存呢？128MB呢？

可伸缩Buffer（Resizable Buffers）

另一个方案是在每个Message Reader内部维护一个容量可变的buffer。一个可变的buffer在初始化时占用较少控件，在消息变得很大超出容量时自动扩容。这样每个链接就不需要都占用比如1MB的空间。每个链接只使用承载下一个消息所必须的内存大小。

要实现一个可伸缩的buffer有几种不同的办法。每一种都有它的优缺点，下面几个小结我会逐一讨论它们。

拷贝扩容（Resize by Copy）

第一种实现可伸缩buffer的办法是初始化buffer的时候只申请较少的空间，比如4KB。如果消息超出了4KB的大小那么开赔一个更大的空间，比如8KB，然后把4KB中的数据拷贝纸8KB的内存块中。

以拷贝方式扩容的优点是一个消息的全部数据都被保存在了一个连续的字节数组中。这使得数据解析变得更加容易。

同时它的缺点是会增加大量的数据拷贝操作。

为了减少数据的拷贝操作，你可以分析整个消息流中的消息大小，一次来找到最适合当前机器的可以减少拷贝操作的buffer大小。例如，你可能会注意到觉大多数的消息都是小于4KB的，因为他们仅仅包含了一个非常请求和响应。这意味着消息的处所荣校应该设置为4KB。

同时，你可能会发现如果一个消息大于4KB，很可能是因为他包含了一个文件。你会可能注意到 大多数通过系统的数据都是小于128KB的。所以我们可以在第一次扩容设置为128KB。

最后你可能会发现当一个消息大于128KB后，没有什么规律可循来确定下次分配的空间大小，这意味着最后的buffer容量应该设置为消息最大的可能数据量。

结合这三次扩容时的大小设置，可以一定程度上减少数据拷贝。4KB以下的数据无需拷贝。在1百万的连接下需要的空间例如1,000,000x4KB=4GB，目前（2015）大多数服务器都扛得住。4KB到128KB会仅需拷贝一次，即拷贝4KB数据到128KB的里面。消息大小介于128KB和最大容量的时需要拷贝两次。首先4KB数据被拷贝第二次是拷贝128KB的数据，所以总共需要拷贝132KB数据。假设没有很多的消息会超过128KB，那么这个方案还是可以接受的。

当一个消息被完整的处理完毕后，它占用的内容应当即刻被释放。这样下一个来自东一个链接通道的消息可以从最小的buffer大小重新开始。这个操作是必须的如果我们需要尽可能高效地复用不同链接之间的内存。大多数情况下并不是所有的链接都会在同一时刻需要大容量的buffer。

笔者写了一个完整的教程阐述了如何实现一个内存buffer使其支持扩容：Resizable Arrays 。这个教程也附带了一个指向GitHub上的源码仓地址，里面有实现方案的具体代码。

追加扩容（Resize by Append）

另一种实现buffer扩容的方案是让buffer包含几个数组。当需要扩容的时候只需要在开辟一个新的字节数组，然后把内容写到里面去。

这种扩容也有两个具体的办法。一中是开辟单独的字节数组，然后用一个列表把这些独立数组关联起来。另一种是开辟一些更大的，相互共享的字节数组切片，然后用列表把这些切片和buffer关联起来。个人而言，笔者认为第二种切片方案更好一点点，但是它们之前的差异比较小。（译者话：关于这两个办法，我个人觉得概念介绍有点难懂，建议读者也参考一下原文😏）

这种追加扩容的方案不管是用独立数组还是切片都有一个优点，那就是写数据的时候不需要二外的拷贝操作。所有的数据可以直接从socket（Channel）中拷贝至数组活切片当中。

这种方案的缺点也很明显，就是数据不是存储在一个连续的数组中。这会使得数据的解析变得更加复杂，因为解析器不得不同时查找每一个独立数组的结尾和所有数组的结尾。正因为我们需要在写数据时查找消息的结尾，这个模型在设计实现时会相对不那么容易。

TLV编码消息(TLV Encoded Messages)

有些协议的消息消失采用的是一种TLV格式（Type, Length, Value）。这意味着当消息到达时，消息的完整大小存储在了消息的开始部分。我们可以立刻判断为消息开辟多少内存空间。

TLV编码是的内存管理变得更加简单。我们可以立刻知道为消息分配多少内存。即便是不完整的消息，buffer结尾后面也不会有浪费的内存。

TLV编码的一个缺点是我们需要在消息的全部数据接收到之前就开辟好需要用的所有内存。因此少量链接慢，但发送了大块数据的链接会占用较多内存，导致服务器无响应。

解决上诉问题的一个变通办法是使用一种内部包含多个TLV的消息格式。这样我们为每个TLV段分配内存而不是为整个的消息分配，并且只在消息的片段到达时才分配内存。但是消息片段很大时，任然会出现一样的问题。

另一个办法是为消息设置超时，如果长时间未接收到的消息（比如10-15秒）。这可以让服务器从偶发的并发处理大块消息恢复过来，不过还是会让服务器有一段时间无响应。另外恶意的DoS攻击会导致服务器开辟大量内存。

TLV编码有不同的变种。有多少字节使用这样确切的类型和字段长度取决于每个独立的TLV编码。有的TLV编码吧字段长度放在前面，接着放类型，最后放值。尽管字段的顺序不同，但他任然是一个TLV的类型。

TLV编码使得内存管理更加简单，这也是HTTP1.1协议让人觉得是一个不太优良的的协议的原因。正因如此，HTTP2.0协议在设计中也利用TLV编码来传输数据帧。也是因为这个原因我们设计了自己的利用TLV编码的网络协议VStack.co。

写不完整的消息（**Writing Partial Messages**）

在非阻塞IO管道中，写数据也是一个不小的挑战。当你调用一个非阻塞模式Channel的write()方法时，无法保证有多少机字节被写入了ByteBuffer中。write方法返回了实际写入的字节数，所以跟踪记录已被写入的字节数也是可行的。这就是我们遇到的问题：持续记录被写入的不完整的小树知道一个消息中所有的数据都发送完毕。

为了管理不完整消息的写操作，我们需要创建一个Message Writer。正如前面的Message Reader，我们也需要每个Channel配备一个Message Writer来写数据。在每个Message Writer中我们记录准确的已经写入的字节数。

为了避免多个消息传递到Message Writer超出他所能处理到Channel的量，我们需要让到达的消息进入队列。Message Writer则尽可能快的将数据写到Channel里。

下面是一个流程图，展示的是不完整消息被写入的过程：

为了使Message Writer能够持续发送刚才已经发送了一部分的消息，Message Writer需要被移植调用，这样他就可以发送更多数据。

如果你有大量的链接，你会持有大量的Message Writer实例。检查比如1百万的Message Writer实例是来确定他们是否处于可写状态是很慢的操作。首先，许多Message Writer可能根本就没有数据需要发送。我们不想检查这些实例。其次，不是所有的Channel都处于可写状

态。我们不想浪费时间在这些非写入状态的Channel。

为了检查一个Channel是否可写，可以把它注册到Selector上。但是我们不希望把所有的Channel实例都注册到Selector。试想一下，如果你有1百万的链接，这里面大部分是空闲的，把1百万链接都祖册到Selector上。然后调用select方法的时候就会有很多的Channel处于可写状态。你需要检查所有这些链接中的Message Writer以确认是否有数据可写。

为了避免检查所有的这些Message Writer，以及那些根本没有消息需要发送给他们的Channel实例，我么可以采用入校两步策略：

1. 当有消息写入到Message Writer忠厚，把它关联的Channel注册到Selector上（如果还未注册的话）。

2. 当服务器有空的时候，可以检查Selector看看注册在上面的Channel实例是否处于可写状态。每个可写的channel，使其Message Writer向Channel中写入数据。如果Message Writer已经把所有的消息都写入Channel，把Channel从Selector上解绑。

这两个小步骤确保只有有数据要写的Channel才会被注册到Selector。

集成（Putting it All Together）

正如你所知到的，一个被阻塞的服务器需要时刻检查当前是否有显得完整消息抵达。在一个消息被完整的收到前，服务器可能需要检查多次。检查一次是不够的。

类似的，服务器也需要时刻检查当前是否有任何可写的数据。如果有的话，服务器需要检查相应的链接看他们是否处于可写状态。仅仅在消息第一次进入队列时检查是不够的，因为一个消息可能被部分写入。

总而言之，一个非阻塞的服务器要三个管道，并且经常执行：

- 读数据管道，用来检查打开的链接是否有新的数据到达；
- 处理数据管道，负责处理接收到的完整消息；
- 写数据管道，用于检查是否有数据可以写入打开的连接中；

这三个管道在循环中重复执行。你可以尝试优化它的执行。比如，如果没有消息在队列中等候，那么可以跳过写数据管道。或者，如果没有收到新的完整消息，你甚至可以跳过处理数据管道。

下面这张流程图阐述了这整个服务器循环过程：



假如你还是柑橘这比较复杂难懂，可以去clone我们的源码仓：https://github.com/jjenkov/java-nio-server 也许亲眼看到了代码会帮助你理解这一块是如何实现的。

服务器线程模型（Server Thread Model）

我们在GitHub上的源码中实现的非阻塞IO服务使用了一个包含两条线程的线程模型。第一个线程负责从ServerSocketChannel接收到达的链接。另一个线程负责处理这些链接，包括读消息，处理消息，把响应写回到链接。这个双线程模型如下：



前一节中已经介绍过的服务器的循环处理在处理线程中执行。

12. Java NIO DatagramChannel数据报通道

> 原文链接：http://tutorials.jenkov.com/java-nio/datagram-channel.html

- 打开一个DatagramChannel（Opening a DatagramChannel）
- 接收数据（Receiving Data）
- 发送数据（Sending Data）
- 链接特定机器地址（Connecting to a Specific Address）

一个Java NIO DatagramChannel死一个可以发送、接收UDP数据包的通道。由于UDP是面向无连接的网络协议，我们不可用像使用其他通道一样直接进行读写数据。正确的做法是发送、接收数据包。

打开一个DatagramChannel（Opening a DatagramChannel）

打开一个DatagramChannel你这么操作：

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));
```

上述示例中，我们打开了一个DatagramChannel，它可以在9999端口上收发UDP数据包。

接收数据（Receiving Data）

接收数据，直接调用DatagramChannel的receive()方法：

```
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();

**channel.receive(buf);**
```

receive()方法会把接收到的数据包中的数据拷贝至给定的Buffer中。如果数据包的内容超过了Buffer的大小，剩余的数据会被直接丢弃。

发送数据（Sending Data）

发送数据是通过DatagramChannel的send()方法：

```
String newData = "New String to wrte to file..."              +System.currentTimeMill
is();
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();

**int byteSent = channel.send(buf, new InetSocketAddress("jenkov.com", 80));**
```

上述示例会吧一个字符串发送到“jenkov.com”服务器的UDP端口80.目前这个端口没有被任何程序监听，所以什么都不会发生。当发送了数据后，我们不会收到数据包是否被接收的的通知，这是由于UDP本身不保证任何数据的发送问题。

链接特定机器地址（Connecting to a Specific Address）

DatagramChannel实际上是可以指定到网络中的特定地址的。由于UDP是面向无连接的，这种链接方式并不会创建实际的连接，这和TCP通道类似。确切的说，他会锁定DatagramChannel,这样我们就只能通过特定的地址来收发数据包。

看一个例子先：

```
channel.connect(new InetSocketAddress("jenkov.com"), 80));
```

当连接上后，可以向使用传统的通道那样调用read()和Writer()方法。区别是数据的读写情况得不到保证。下面是几个示例：

```
int bytesRead = channel.read(buf);
```

```
int bytesWritten = channel.write(buf);
```

13.Java NIO Pipe管道

> 原文链接：http://tutorials.jenkov.com/java-nio/pipe.html

一个Java NIO的管道是两个线程间单向传输数据的连接。一个管道（Pipe）有一个source channel和一个sink channel(没想到合适的中文名)。我们把数据写到sink channel中，这些数据可以同过source channel再读取出来。

下面是一个管道的示意图：



创建管道(Creating a Pipe)

打开一个管道通过调用Pipe.open()工厂方法，如下：

```
Pipe pipe = Pipe.open();
```

向管道写入数据（Writing to a Pipe）

向管道写入数据需要访问他的sink channel：

```
Pipe.SinkChannel sinkChannel = pipe.sink();
```

接下来就是调用write()方法写入数据了：

```
String newData = "New String to write to file..." + System.currentTimeMillis();

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());

buf.flip();

while(buf.hasRemaining()) {
    sinkChannel.write(buf);
}
```

从管道读取数据（Reading from a Pipe）

类似的从管道中读取数据需要访问他的source channel：

```
Pipe.SourceChannel sourceChannel = pipe.source();
```

接下来调用read()方法读取数据：

```
ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf);
```

注意这里read()的整形返回值代表实际读取到的字节数。

14. Java NIO vs. IO

> 原文链接：http://tutorials.jenkov.com/java-nio/nio-vs-io.html

当学习Java的NIO和IO时，有个问题会跳入脑海当中：什么时候该用IO，什么时候用NIO？

下面的章节中笔者会试着分享一些线索，包括两者之间的区别，使用场景以及他们是如何影响代码设计的。

NIO和IO之间的主要差异（Mian Differences Between Java NIO and IO）

下面这个表格概括了NIO和IO的主要差异。我们会针对每个差异进行解释。

IO	NIO
Stream oriented	Buffer oriented
Blocking IO	No blocking IO
	Selectors

面向流和面向缓冲区比较(Stream Oriented vs. Buffer Oriented)

第一个重大差异是IO是面向流的，而NIO是面向缓存区的。这句话是什么意思呢？

Java IO面向流意思是我们每次从流当中读取一个或多个字节。怎么处理读取到的字节是我们自己的事情。他们不会再任何地方缓存。再有就是我们不能在流数据中向前后移动。如果需要向前后移动读取位置，那么我们需要首先为它创建一个缓存区。

Java NIO是面向缓冲区的，这有些细微差异。数据是被读取到缓存当中以便后续加工。我们可以在缓存中向向后移动。这个特性给我们处理数据提供了更大的弹性空间。当然我们任然需要在使用数据前检查缓存中是否包含我们需要的所有数据。另外需要确保在往缓存中写入数据时避免覆盖了已经写入但是还未被处理的数据。

阻塞和非阻塞IO比较（Blocking vs. No-blocking IO）

Java IO的各种流都是阻塞的。这意味着一个线程一旦调用了read(),write()方法，那么该线程就被阻塞住了，知道读取到数据或者数据完整写入了。在此期间线程不能做其他任何事情。

Java NIO的非阻塞模式使得线程可以通过channel来读数据，并且是返回当前已有的数据，或者什么都不返回如果但钱没有数据可读的话。这样一来线程不会被阻塞住，它可以继续向下执行。

通常线程在调用非阻塞操作后，会通知处理其他channel上的IO操作。因此一个线程可以管理多个channel的输入输出。

Selectors

Java NIO的selector允许一个单一线程监听多个channel输入。我们可以注册多个channel到selector上，然后然后用一个线程来挑出一个处于可读或者可写状态的channel。selector机制使得单线程管理过个channel变得容易。

NIO和IO是如何影响程序设计的（How NIO and IO Influences Application Design）

开发中选择NIO或者IO会在多方面影响程序设计：

1. 使用NIO、IO的API调用类
2. 数据处理
3. 处理数据需要的线程数

API调用(The API Calls)

显而易见使用NIO的API接口和使用IO时是不同的。不同于直接冲InputStream读取字节，我们的数据需要先写入到buffer中，然后再从buffer中处理它们。

数据处理（The Processing of Data）

数据的处理方式也随着是NIO或IO而异。 在IO设计中，我们从InputStream或者Reader中读取字节。假设我们现在需要处理一个按行排列的文本数据，如下：

```
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890
```

这个处理文本行的过程大概是这样的：

```
InputStream input = ... ; // get the InputStream from the client socket

BufferedReader reader = new BufferedReader(new InputStreamReader(input));

String nameLine   = reader.readLine();
String ageLine    = reader.readLine();
String emailLine  = reader.readLine();
String phoneLine  = reader.readLine();
```

小结

NIO允许我们只用一条线程来管理多个通道（网络连接或文件），随之而来的代价是解析数据相对于阻塞流来说可能会变得更加的复杂。

如果你需要同时管理成千上万的链接，这些链接只发送少量数据，例如聊天服务器，用NIO来实现这个服务器是有优势的。类似的，如果你需要维持大量的链接，例如P2P网络，用单线程来管理这些 链接也是有优势的。这种单线程多连接的设计可以用下图描述：



Java NIO: A single thread managing multiple connections

如果链接数不是很多，但是每个链接的占用较大带宽，每次都要发送大量数据，那么使用传统的IO设计服务器可能是最好的选择。下面是经典IO服务设计图：



Java IO: A classic IO server design - one connection handled by one thread.

15.Java NIO Path路径

> 原文链接：http://tutorials.jenkov.com/java-nio/path.html

Java的path接口是作为Java NIO 2的一部分是Java6,7中NIO的升级增加部分。Path在Java 7新增的。相关接口位于java.nio.file包下，所以Javaz内Path接口的完整名称是java.nio.file.Path.

一个Path实例代表一个文件系统内的路径。path可以指向文件也可以指向目录。可以使相对路径也可以是绝对路径。绝对路径包含了从根目录到该文件（目录）的完整路径。相对路径包含该文件（目录）相对于其他路径的路径。相对路径听起来可能有点让人头晕。但是别急，稍后我们会详细介绍。

不要把文件系统中路径和环境变量的路径混淆。java.nio.file.Path和环境变量没有任何关系。

在很多情况下java.no.file.Path接口和java.io.File比较相似，但是他们之间存在一些细微的差异。尽管如此，在大多数情况下，我们都可以用File相关类来替换Path接口。

创建Path实例（Creating a Path Instance）

为了使用java.nio.file.Path实例我们必须创建Path对象。创建Path实例可以通过Paths的工厂方法get（）。下面是一个实例：

```
import java.nio.file.Path;
import java.nio.file.Paths;

public classs PathExample {
  public static void mian(String[] args) {
    Path = path = Paths.get("c:\\data\\myfile.txt");
  }
}
```

注意上面的两个import声明。需要使用Path和Paths的接口,毕现先把他们引入。

其次注意Paths.get("c:\data\myfile.txt")的调用。这个方法会创建一个Path实例，换句话说Paths.get()是Paths的一个工厂方法。

创建绝对路径（Creating an Absolute Path）

创建绝对路径只需要调动Paths.get()这个工厂方法，同时传入绝对文件。这是一个例子：

```
Path path = Paths.get("c:\\data\\myfile.txt");
```

对路径是c:\data\myfile.txt，里面的双斜杠\字符是Java 字符串中必须的，因为\是转义字符，表示后面跟的字符在字符串中的真实含义。双斜杠\表示\自身。

上面的路径是Windows下的文件系统路径表示。在Unixx系统中（Linux, MacOS,FreeBSD等）上述的绝对路径长得是这样的：

```
Path path = Paths.get("/home/jakobjenkov/myfile.txt");
```

他的绝对路径是/home/jakobjenkov/myfile.txt。 如果在Windows机器上使用用这种路径，那么这个路径会被认为是相对于当前磁盘的。例如：

```
/home/jakobjenkov/myfile.txt
```

这个路径会被理解其C盘上的文件，所以路径又变成了

```
C:/home/jakobjenkov/myfile.txt
```

创建相对路径（Creating a Relative Path）

相对路径是从一个路径（基准路径）指向另一个目录或文件的路径。完整路径实际上等同于相对路径加上基准路径。

Java NIO的Path类可以用于相对路径。创建一个相对路径可以通过调用Path.get(basePath, relativePath),下面是一个示例：

```
Path projects = Paths.get("d:\\data", "projects");

Path file     = Paths.get("d:\\data", "projects\\a-project\\myfile.txt");
```

第一行创建了一个指向d:\data\projects的Path实例。第二行创建了一个指向d:\data\projects\a-project\myfile.txt的Path实例。 在使用相对路径的时候有两个特殊的符号：

- .
- ..

.表示的是当前目录，例如我们可以这样创建一个相对路径：

```
Path currentDir = Paths.get(".");
System.out.println(currentDir.toAbsolutePath());
```

currentDir的实际路径就是当前代码执行的目录。 如果在路径中间使用了.那么他的含义实际上就是目录位置自身，例如：

```
Path currentDir = Paths.get("d:\\data\\projects\.\a-project");
```

上诉路径等同于：

```
d:\data\projects\a-project
```

..表示父目录或者说是上一级目录：

```
Path parentDir = Paths.get("..");
```

这个Path实例指向的目录是当前程序代码的父目录。 如果在路径中间使用..那么会相应的改变指定的位置：

```
String path = "d:\\data\\projects\\a-project\\..\\another-project";
Path parentDir2 = Paths.get(path);
```

这个路径等同于：

```
d:\data\projects\another-project
```

.和..也可以结合起来用，这里不过多介绍。

Path.normalize()

Path的normalize()方法可以把路径规范化。也就是把.和..都等价去除：

```
String originalPath = "d:\\data\\projects\\a-project\\..\\another-project";

Path path1 = Paths.get(originalPath);
System.out.println("path1 = " + path1);

Path path2 = path1.normalize();
System.out.println("path2 = " + path2);
```

这段代码的输出如下：

```
path1 = d:\data\projects\a-project\..\another-project
path2 = d:\data\projects\another-project
```

16. Java NIO Files

> 原文链接：http://tutorials.jenkov.com/java-nio/files.html

Java NIO中的Files类（java.nio.file.Files）提供了多种操作文件系统中文件的方法。本节教程将覆盖大部分方法。Files类包含了很多方法，所以如果本文没有提到的你也可以直接查询JavaDoc文档。

java.nio.file.Files类是和java.nio.file.Path相结合使用的，所以在用Files之前确保你已经理解了Path类。

Files.exists()

Files.exits()方法用来检查给定的Path在文件系统中是否存在。 在文件系统中创建一个原本不存在的Payh是可行的。例如，你想新建一个目录，那么闲创建对应的Path实例，然后创建目录。

由于Path实例可能指向文件系统中的不存在的路径，所以需要用Files.exists()来确认。

下面是一个使用Files.exists()的示例：

```
Path path = Paths.get("data/logging.properties");

boolean pathExists =
        Files.exists(path,
            new LinkOption[]{ LinkOption.NOFOLLOW_LINKS});
```

这个示例中，我们首先创建了一个Path对象，然后利用Files.exists()来检查这个路径是否真实存在。

注意Files.exists()的的第二个参数。他是一个数组，这个参数直接影响到Files.exists()如何确定一个路径是否存在。在本例中，这个数组内包含了LinkOptions.NOFOLLOW_LINKS，表示检测时不包含符号链接文件。

Files.createDirectory()

Files.createDirectory()会创建Path表示的路径，下面是一个示例：

```
Path path = Paths.get("data/subdir");

try {
    Path newDir = Files.createDirectory(path);
} catch(FileAlreadyExistsException e){
    // the directory already exists.
} catch (IOException e) {
    //something else went wrong
    e.printStackTrace();
}
```

第一行创建了一个Path实例，表示需要创建的目录。接着用try-catch把Files.createDirectory()的调用捕获住。如果创建成功，那么返回值就是新创建的路径。

如果目录已经存在了，那么会抛出java.nio.file.FileAlreadyExistException异常。如果出现其他问题，会抛出一个IOException。比如说，要创建的目录的父目录不存在，那么就会抛出IOException。父目录指的是你要创建的目录所在的位置。也就是新创建的目录的上一级父目录。

Files.copy()

Files.copy()方法可以吧一个文件从一个地址复制到另一个位置。例如：

```
Path sourcePath      = Paths.get("data/logging.properties");
Path destinationPath = Paths.get("data/logging-copy.properties");

try {
    Files.copy(sourcePath, destinationPath);
} catch(FileAlreadyExistsException e) {
    //destination file already exists
} catch (IOException e) {
    //something else went wrong
    e.printStackTrace();
}
```

这个例子当中，首先创建了原文件和目标文件的Path实例。然后把它们作为参数，传递给Files.copy(),接着就会进行文件拷贝。

如果目标文件已经存在，就会抛出java.nio.file.FileAlreadyExistsException异常。类似的吐过中间出错了，也会抛出IOException。

覆盖已经存在的文件(Overwriting Existing Files)

copy操作可以强制覆盖已经存在的目标文件。下面是具体的示例：

```
Path sourcePath      = Paths.get("data/logging.properties");
Path destinationPath = Paths.get("data/logging-copy.properties");

try {
    Files.copy(sourcePath, destinationPath,
            StandardCopyOption.REPLACE_EXISTING);
} catch(FileAlreadyExistsException e) {
    //destination file already exists
} catch (IOException e) {
    //something else went wrong
    e.printStackTrace();
}
```

注意copy方法的第三个参数，这个参数决定了是否可以覆盖文件。

Files.move()

Java NIO的Files类也包含了移动的文件的接口。移动文件和重命名是一样的，但是还会改变文件的目录位置。java.io.File类中的renameTo()方法与之功能是一样的。

```
Path sourcePath      = Paths.get("data/logging-copy.properties");
Path destinationPath = Paths.get("data/subdir/logging-moved.properties");

try {
    Files.move(sourcePath, destinationPath,
            StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) {
    //moving file failed.
    e.printStackTrace();
}
```

首先创建源路径和目标路径的，原路径指的是需要移动的文件的初始路径，目标路径是指需要移动到的位置。

这里move的第三个参数也允许我们覆盖已有的文件。

Files.delete()

Files.delete()方法可以删除一个文件或目录：

```
Path path = Paths.get("data/subdir/logging-moved.properties");

try {
    Files.delete(path);
} catch (IOException e) {
    //deleting file failed
    e.printStackTrace();
}
```

首先创建需要删除的文件的path对象。接着就可以调用delete了。

Files.walkFileTree()

Files.walkFileTree()方法具有递归遍历目录的功能。walkFileTree接受一个Path和FileVisitor作为参数。Path对象是需要遍历的目录，FileVistor则会在每次遍历中被调用。

下面先来看一下FileVisitor这个接口的定义：

```
public interface FileVisitor {

    public FileVisitResult preVisitDirectory(
        Path dir, BasicFileAttributes attrs) throws IOException;

    public FileVisitResult visitFile(
        Path file, BasicFileAttributes attrs) throws IOException;

    public FileVisitResult visitFileFailed(
        Path file, IOException exc) throws IOException;

    public FileVisitResult postVisitDirectory(
        Path dir, IOException exc) throws IOException {

}
```

FileVisitor需要调用方自行实现，然后作为参数传入walkFileTree().FileVisitor的每个方法会在遍历过程中被调用多次。如果不需要处理每个方法，那么可以继承他的默认实现类SimpleFileVisitor，它将所有的接口做了空实现。

下面看一个walkFileTree()的示例：

```
Files.walkFileTree(path, new FileVisitor<Path>() {
  @Override
  public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws
 IOException {
    System.out.println("pre visit dir:" + dir);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOExce
ption {
    System.out.println("visit file: " + file);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult visitFileFailed(Path file, IOException exc) throws IOExceptio
n {
    System.out.println("visit file failed: " + file);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOExcept
ion {
    System.out.println("post visit directory: " + dir);
    return FileVisitResult.CONTINUE;
  }
});
```

FileVisitor的方法会在不同时机被调用： preVisitDirectory()在访问目录前被调用。postVisitDirectory()在访问后调用。

visitFile()会在整个遍历过程中的每次访问文件都被调用。他不是针对目录的，而是针对文件的。visitFileFailed()调用则是在文件访问失败的时候。例如，当缺少合适的权限或者其他错误。

上述四个方法都返回一个FileVisitResult枚举对象。具体的可选枚举项包括：

- CONTINUE
- TERMINATE
- SKIP_SIBLINGS
- SKIP_SUBTREE

返回这个枚举值可以让调用方决定文件遍历是否需要继续。 CONTINE表示文件遍历和正常情况下一样继续。

TERMINATE表示文件访问需要终止。

SKIP_SIBLINGS表示文件访问继续，但是不需要访问其他同级文件或目录。

SKIP_SUBTREE表示继续访问，但是不需要访问该目录下的子目录。这个枚举值仅在preVisitDirectory()中返回才有效。如果在另外几个方法中返回，那么会被理解为CONTINE。

Searching For Files

下面看一个例子，我们通过walkFileTree()来寻找一个README.txt文件：

```
Path rootPath = Paths.get("data");
String fileToFind = File.separator + "README.txt";

try {
  Files.walkFileTree(rootPath, new SimpleFileVisitor<Path>() {

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOEx
ception {
      String fileString = file.toAbsolutePath().toString();
      //System.out.println("pathString = " + fileString);

      if(fileString.endsWith(fileToFind)){
        System.out.println("file found at path: " + file.toAbsolutePath());
        return FileVisitResult.TERMINATE;
      }
      return FileVisitResult.CONTINUE;
    }
  });
} catch(IOException e){
    e.printStackTrace();
}
```

Deleting Directies Recursively

Files.walkFileTree()也可以用来删除一个目录以及内部的所有文件和子目。Files.delete()只用用于删除一个空目录。我们通过遍历目录，然后在visitFile()接口中三次所有文件，最后在postVisitDirectory()内删除目录本身。

```
Path rootPath = Paths.get("data/to-delete");

try {
  Files.walkFileTree(rootPath, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOEx
ception {
      System.out.println("delete file: " + file.toString());
      Files.delete(file);
      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOExce
ption {
      Files.delete(dir);
      System.out.println("delete dir: " + dir.toString());
      return FileVisitResult.CONTINUE;
    }
  });
} catch(IOException e){
  e.printStackTrace();
}
```

Additional Methods in the Files Class

java.nio.file.Files类还有其他一些很有用的方法，比如创建符号链接，确定文件大小以及设置文件权限等。具体用法可以查阅JavaDoc中的API说明。

17. Java NIO AsynchronousFileChannel异步文件通道

> 原文链接：http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html

Java7中新增了AsynchronousFileChannel作为nio的一部分。AsynchronousFileChannel使得数据可以进行异步读写。下面将介绍一下AsynchronousFileChannel的使用。

创建AsynchronousFileChannel（Creating an AsynchronousFileChannel）

AsynchronousFileChannel的创建可以通过open()静态方法：

```
Path path = Paths.get("data/test.xml");

AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.READ);
```

open()的第一个参数是一个Path实体，指向我们需要操作的文件。 第二个参数是操作类型。上述示例中我们用的是StandardOpenOption.READ，表示以读的形式操作文件。

读取数据（Reading Data）

读取AsynchronousFileChannel的数据有两种方式。每种方法都会调用AsynchronousFileChannel的一个read()接口。下面分别看一下这两种写法。

通过Future读取数据（Reading Data Via a Future）

第一种方式是调用返回值为Future的read()方法：

```
Future<Integer> operation = fileChannel.read(buffer, 0);
```

这种方式中，read()接受一个ByteBuffer座位第一个参数，数据会被读取到ByteBuffer中。第二个参数是开始读取数据的位置。

read()方法会立刻返回，即使读操作没有完成。我们可以通过isDone()方法检查操作是否完成。

下面是一个略长的示例：

```
 AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.READ);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

Future<Integer> operation = fileChannel.read(buffer, position);

while(!operation.isDone());

buffer.flip();
byte[] data = new byte[buffer.limit()];
buffer.get(data);
System.out.println(new String(data));
buffer.clear();
```

在这个例子中我们创建了一个AsynchronousFileChannel，然后创建一个ByteBuffer作为参数传给read。接着我们创建了一个循环来检查是否读取完毕isDone()。这里的循环操作比较低效，它的意思是我们需要等待读取动作完成。

一旦读取完成后，我们就可以把数据写入ByteBuffer，然后输出。

通过CompletionHandler读取数据（Reading Data Via a CompletionHandler）

另一种方式是调用接收CompletionHandler作为参数的read()方法。下面是具体的使用：

```
fileChannel.read(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>(
) {
    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        System.out.println("result = " + result);

        attachment.flip();
        byte[] data = new byte[attachment.limit()];
        attachment.get(data);
        System.out.println(new String(data));
        attachment.clear();
    }

    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {

    }
});
```

这里，一旦读取完成，将会触发CompletionHandler的completed()方法，并传入一个Integer和ByteBuffer。前面的整形表示的是读取到的字节数大小。第二个ByteBuffer也可以换成其他合适的对象方便数据写入。 如果读取操作失败了，那么会触发failed()方法。

写数据（Writing Data）

和读数据类似某些数据也有两种方式，调动不同的的write()方法，下面分别看介绍这两种方法。

通过Future写数据（Writing Data Via a Future）

通过AsynchronousFileChannel我们可以一步写数据

```
Path path = Paths.get("data/test-write.txt");
AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

buffer.put("test data".getBytes());
buffer.flip();

Future<Integer> operation = fileChannel.write(buffer, position);
buffer.clear();

while(!operation.isDone());

System.out.println("Write done");
```

首先把文件已写方式打开，接着创建一个ByteBuffer座位写入数据的目的地。再把数据进入ByteBuffer。最后检查一下是否写入完成。 需要注意的是，这里的文件必须是已经存在的，否者在尝试write数据是会抛出一个java.nio.file.NoSuchFileException.

检查一个文件是否存在可以通过下面的方法：

```
if(!Files.exists(path)){
    Files.createFile(path);
}
```

通过CompletionHandler写数据（Writing Data Via a CompletionHandler）

我们也可以通过CompletionHandler来写数据：

```
Path path = Paths.get("data/test-write.txt");
if(!Files.exists(path)){
    Files.createFile(path);
}
AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

buffer.put("test data".getBytes());
buffer.flip();

fileChannel.write(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>
() {

    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        System.out.println("bytes written: " + result);
    }

    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {
        System.out.println("Write failed");
        exc.printStackTrace();
    }
});
```

同样当数据吸入完成后completed()会被调用，如果失败了那么failed()会被调用。